Análise de Complexidade

Sergio Canuto
sergio.canuto@ifg.edu.br

Avisos

Próxima avaliação:

01/12

AVALIAÇÃO SUBSTITUTIVA:

08/12

Análise e Complexidade de Algoritmos

- A análise de algoritmos busca responder à seguinte pergunta: podemos fazer um algoritmo mais eficiente?
- Algoritmos diferentes, mas capazes de resolver o mesmo problema, não necessariamente o fazem com a mesma eficiência.
  - Essas diferenças de eficiência podem:
    - **Ser irrelevantes** para um pequeno número de elementos processados.
    - **Crescer proporcionalmente** com o número de elementos processados.
- Para comparar a eficiência dos algoritmos foi criada uma medida chamada **complexidade computacional**.

Análise e Complexidade de Algoritmos

- Para determinar se um algoritmo é o mais eficiente, podemos utilizar duas abordagens:
  - **Análise empírica**: comparação entre os programas.
  - **Análise matemática**: estudo das propriedades do algoritmo.

Análise e Complexidade de Algoritmos - Análise Empírica

- Avalia o custo (ou a complexidade) de um algoritmo a partir da avaliação de sua execução quando implementado.
- Na **análise empírica**, um algoritmo é analisado pela execução de seu programa correspondente.
  - Ex.: Quanto tempo demorou? Quanto de memória gastou?
- A análise empírica possui uma série de vantagens. Por meio dela podemos:
  - Avaliar o desempenho em determinada configuração de computador/linguagem.
  - Considerar custos não aparentes. Por exemplo, o custo da alocação de memória.
  - Comparar computadores.
  - Comparar linguagens.

Análise e Complexidade de Algoritmos - Análise Empírica

- Apesar de suas vantagens, existem certas dificuldades na realização da análise empírica
  - Necessidade de implementar o algoritmo. Isso depende da habilidade do programador.
  - Resultado pode ser mascarado pelo hardware (computador utilizado) ou software (eventos ocorridos no momento de avaliação).
  - Qual a natureza dos dados: reais, aleatórios ou perversos?
    - O uso de **dados aleatórios** permite avaliar o desempenho médio do algoritmo. **Dados perversos** permitem avaliar o desempenho no pior caso.

Análise e Complexidade de Algoritmos - ANÁLISE MATEMÁTICA

- Muitas vezes, é preferível que a medição do tempo gasto por um algoritmo seja feita de maneira independente do hardware ou da linguagem usada na sua implementação. Nesse tipo de situação, convém utilizar a análise matemática do algoritmo.
- Nessa análise, estamos avaliando a **ideia** por trás do algoritmo.
  - Para tanto, detalhes de baixo nível, como a linguagem de programação utilizada, o hardware no qual o algoritmo é executado ou o conjunto de instruções da CPU são ignorados.
  - Esse tipo de análise permite entender como um algoritmo se comporta **à medida que o conjunto de dados de entrada cresce.**
  - Expressa a relação entre o conjunto de dados de entrada e a quantidade de tempo necessária para processar esses dados.

ANÁLISE MATEMÁTICA - Contando instruções

- Para entender como calcular o custo de um algoritmo, veja o exemplo abaixo.
    - Este algoritmo procura o maior valor presente em um array **A** contendo **n** elementos e o armazena na variável **M**.
- De posse deste trecho de código vamos contar quantas **instruções simples** ele executa.

Exemplo de código

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

ANÁLISE MATEMÁTICA - Contando instruções

- No nosso trecho de código podemos encontrar os seguintes tipos de instruções:
  - Atribuição de um valor a uma variável.
  - Acesso ao valor de um determinado elemento do array.
  - Comparação de dois valores.
  - Incremento de um valor.
  - Operações aritméticas básicas, como adição

Exemplo de código

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

ANÁLISE MATEMÁTICA - Contando instruções

- O custo da **linha 1** é de: 1 instrução (considerando apenas a atribuição).
- O custo da inicialização do laço **for** (**linha 3**) é de: 2 instruções.
    - **uma atribuição (i = 0)**.
    - **uma comparação (i<n).**

Exemplo de código

```
01  int M = A[0];
02
03  for(i = 0; i < n; i++){
04      if(A[i] >= M) {
05          M = A[i];
06      }
07  }
```

ANÁLISE MATEMÁTICA - Contando instruções

- O custo para executar o comando de laço **for** (**linha 3**) é de: 2n instruções.
    - Ao final de cada iteração do laço **for**, precisamos executar mais duas instruções:
        - uma de incremento (**i**++) e
        - uma comparação para verificar se vamos continuar no laço **for** (**i** < **n**).
- No nosso algoritmo, o comando de laço **for** será executado **n** vezes, que é o número de elementos no array **A**. Assim, essas duas instruções também serão executadas **n** vezes, ou seja, o seu custo será $2n$ instruções.
- Se ignorarmos os comandos contidos no corpo do laço **for**, teremos que o algoritmo precisa executar $3 + 2n$ instruções.
    - $f(n) = 2n + 3$.
        - f(n) é a **Função de Custo!**

Exemplo de código

```
01  int M = A[0];
02
03  for(i = 0; i < n; i++){
04      if(A[i] >= M) {
05          M = A[i];
06      }
07  }
```

ANÁLISE MATEMÁTICA - Contando instruções

- Se desconsiderarmos os comandos no corpo do laço **for**, a análise o algoritmo possui um custo de $3 + 2n$ instruções
  - As instruções vistas até o momento são sempre executadas. Porém, as instruções dentro do **for** podem ou não ser executadas.
- Vamos então contar as instruções restantes.
  - Dentro do laço **for** temos um comando de seleção (**if**). Seu custo será de 1 instrução: uma única instrução será responsável pelo acesso ao valor do array e sua comparação (**A[i] >= M**).
  - Dentro do comando **if** temos mais uma instrução: aquela que acessa o valor do array e o atribui a outra variável (**M = A[i]**).
    - No entanto, essa instrução pode ou não ser executada
    - Isso complica um pouco o cálculo do custo do algoritmo.

Exemplo de código

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

ANÁLISE MATEMÁTICA - Contando instruções

- Antes, bastava saber o tamanho do array, **n**, para definir a função de custo f(*n*). Agora, temos que considerar também o conteúdo do array.
- Tome como exemplo dois arrays de mesmo tamanho
    - A1 = {1, 2, 3, 4}
    - A2 = {4, 3, 2, 1}
    - É fácil perceber que o array A1 irá precisar de mais instruções (o comando **if** é sempre **verdadeiro**) para achar o maior valor do que o array A2 (o comando **if** é sempre **falso**).
- Ao analisarmos um algoritmo, é muito comum considerarmos o **pior caso** possível, ou seja, aquele em que o maior número de instruções é executado.

Exemplo de código

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

ANÁLISE MATEMÁTICA - Contando instruções

- No nosso algoritmo, o **pior caso** ocorre quando o array possui valores em ordem crescente.
- Nesta situação, o valor de **M** é sempre substituído, o que resulta em um maior número de instruções.
  - Ou seja, no **pior caso**, o laço **for** sempre executa as duas instruções.
  - Assim, teremos que a função custo do algoritmo será $f(n) = 2n + 2n + 3$, ou $\boldsymbol{f(n) = 4n + 3}$**.**
  - Esta função representa o custo do algoritmo em relação ao tamanho do array (**n**) de entrada no **pior caso**.

Exemplo de código

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

ANÁLISE MATEMÁTICA - Contando instruções

- Será que todos os termos da função *f* são necessários para termos uma noção do custo do algoritmo?
    - Não! Podemos descartar certos termos na função e manter apenas os que nos dizem o que acontece com a função quando o tamanho dos dados de entrada (**n**) cresce muito.
- Em $f(n) = 4n + 3$
    - O termo 3 é simplesmente uma **constante de inicialização**, ou seja, não é afetado pelo valor de **n** e pode, portanto, ser descartado:
        - $f(n) = 4n$
    - Constantes que multiplicam o termo **n** da função também devem ser descartadas.
        - $f(n) = n$

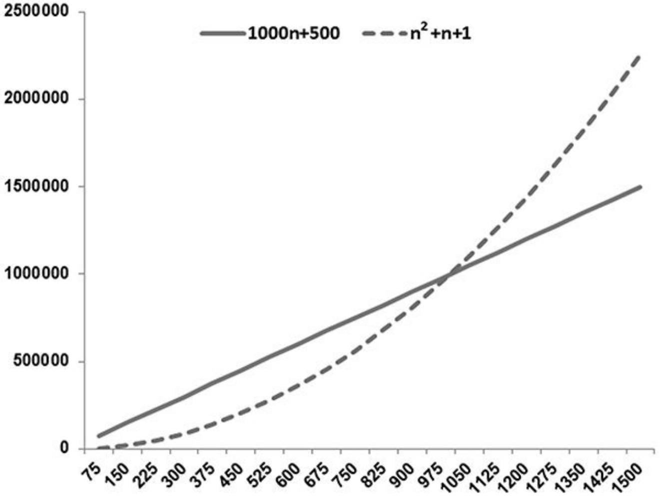
Exemplo de código

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

ANÁLISE MATEMÁTICA - Comportamento Assintótico

- Ao descartarmos todos os termos constantes e manter apenas o de maior crescimento, obtemos o **comportamento assintótico** do algoritmo. Chamamos de comportamento assintótico o comportamento de uma função *f*(*n*) quando *n* tende ao infinito.
- Considere duas funções de custo
  - $g(n) = 1000n + 500$
  - $h(n) = n^2 + n + 1$
- Constantes tornam-se irrelevantes à medida que n cresce.
- Assim, descrevemos a complexidade usando somente o seu custo dominante n para a função $g(n)$ e n^2 para $h(n)$.



ANÁLISE MATEMÁTICA - Comportamento Assintótico

- De modo geral, podemos obter a função de custo de um programa simples apenas analisando as instruções em função da quantidade de laços aninhados.

Função de custo	Comportamento assintótico
$f(n) = 105$	$f(n) = 1$
$f(n) = 15n + 2$	$f(n) = n$
$f(n) = n^2 + 5n + 2$	$f(n) = n^2$
$f(n) = 5n^3 + 200n^2 + 112$	$f(n) = n^3$

Comportamento Assintótico - Notação Grande-O

- Dentre as várias formas de análise assintótica, a mais conhecida e utilizada é a notação **Grande-O** (**O**). Ela representa o custo (seja de tempo ou de espaço) do nosso algoritmo no pior caso possível para todas as entradas de tamanho **n**.
- Ex.: No selection sort, podemos ver dois comandos de laço. Enquanto o laço externo é executado **n** vezes, o número de execuções do laço interno depende do valor do índice do primeiro laço. Assim, o laço interno é executado **n-1** vezes na primeira iteração do laço externo, depois **n-2** vezes, assim por diante.

Método selection sort

```c
void selectionSort(int *V, int n){
    int i, j, me, troca;
    for(i = 0; i < n-1; i++){
        me = i;
        for(j = i+1; j < n; j++){
            if(V[j] < V[me])
                me = j;
        }
        if(i != me){
            troca = V[i];
            V[i] = V[me];
            V[me] = troca;
        }
    }
}
```

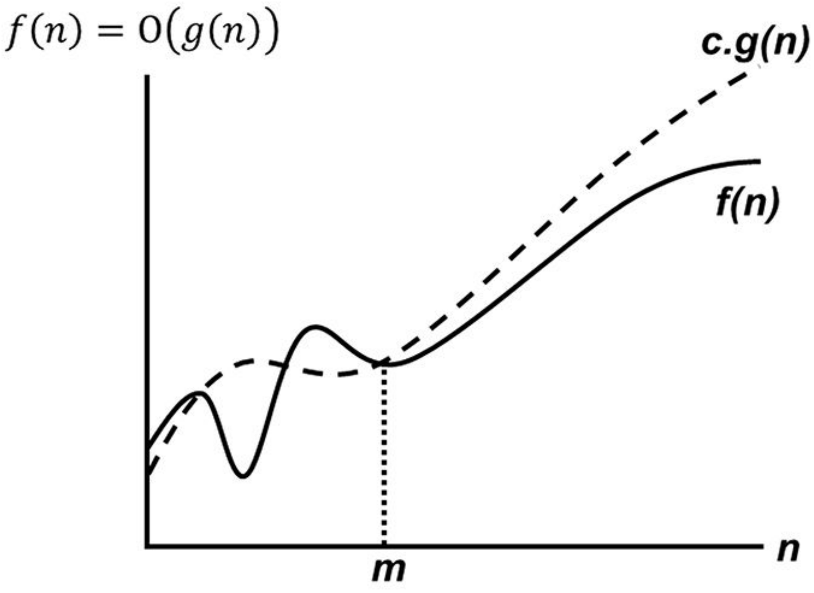
Comportamento Assintótico - Notação Grande-O

- Para calcularmos o custo do **selection sort** temos que calcular o resultado da soma $1 + 2 + ... + (n - 1) + n$. Esta soma representa o número de execuções do laço interno, algo que não é tão simples de se calcular.
- Dependendo do algoritmo, calcular o seu custo exato pode ser uma tarefa muito complicada. No nosso caso, temos que $1 + 2 + ... + (n - 1) + n$ equivale à soma dos n termos de uma **progressão aritmética**, $S(n)$, de razão 1. Assim,
  - $S(n) = 1 + 2 + ... + (n - 1) + n$
  - $S(n) = n(1 + n)/2$

- Uma alternativa mais simples é estimar um **limite superior**.
  - Uma idéia para estimar o limite superior é alterar o algoritmo para algo **menos eficiente** do que temos, e o algoritmo original é, no máximo, tão ruim ou talvez melhor que o novo algoritmo
    - Uma forma de diminuirmos a eficiência do **selection sort** é trocar o laço interno (que muda de tamanho a cada execução do laço externo) por um laço que seja executado sempre **n vezes**. Nesse caso, $f(n) = n^2$
    - A notação Grande-O nesse caso seria $O(n^2)$
    - a notação $O(n^2)$ diz que o custo do algoritmo não é, assintoticamente, pior do que n^2. Em outras palavras, o custo do algoritmo original é, no máximo, tão ruim quanto n^2. Pode ser melhor, mas nunca pior.

Comportamento Assintótico - Notação **Grande *O*** - *Limite Assintótico Superior*

- Matematicamente, a notação O é assim definida: uma função custo $f(n)$ é $O(g(n))$ se existem duas constantes positivas **c** e **m**, tais que, para $n \geq m$, temos $f(n) \leq cg(n)$.
- Em outras palavras, para todos os valores de **n** à direita do valor **m**, o resultado da nossa função custo $f(n)$ é sempre **menor ou igual** ao valor da função usada na notação O, $g(n)$, multiplicada por uma constante **c**.



Exemplo: Comportamento Assintótico - Notação Grande-O

Seja um algoritmo cuja função de custo é $f(n)=3n+3$. Encontre as constantes positivas **c** e **m** para demonstrar que o algoritmo, no pior caso, tem o **limite assintótico superior** $O(n)$.

ex.: https://www.geogebra.org/calculator

Exemplo: Comportamento Assintótico - Notação Grande-O

Seja um algoritmo cuja função de custo é f(n)=3n+3. Encontre as constantes positivas **c** e **m** para demonstrar que o algoritmo, no pior caso, tem o **limite assintótico superior** O(n).

Definição: Uma função de custo $f(n)$ é $O(g(n))$ se existem duas constantes positivas **c** e **m**, tais que, para $n \geq m$, temos $f(n) \leq cg(n)$.

- No nosso caso:
    - f(n)=3n+3
    - g(n)=n

Portanto:

- $f(n) \leq cg(n)$
- $3n+3 \leq cn$.

Vamos escolher c=9. Resolvendo a inequação, $3n+3 \leq 9n$, e portanto, $n \geq 1/2$. Logo, para $n \geq 1/2$ (i.e., m=1/2), a inequação é válida quando escolhemos c=9, por exemplo. Logo, para m=1/2 e c=9, podemos dizer que $f(n) \leq cg(n)$, isto é, $3n+3 \leq cn$. Portanto existem duas constantes c e m que satisfazem a definição, e podemos dizer que a função de custo 3n+3 é O(n)

Exemplo: Comportamento Assintótico - Notação Grande-O

Seja um algoritmo cuja função de custo é f(n)=3n+3. Encontre as constantes positivas **c** e **m** para demonstrar que o algoritmo, no pior caso, tem o **limite assintótico superior** O(n).

Definição: Uma função de custo $f(n)$ é $O(g(n))$ se existem duas constantes positivas **c** e **m**, tais que, para $n \geq m$, temos $f(n) \leq cg(n)$.

- No nosso caso:
  - f(n)=3n+3
  - g(n)=n

Portanto:

  - $f(n) \leq cg(n)$
  - 3n+3 ≤ cn.

Vamos escolher c=9. Resolvendo a inequação, 3n+3≤ 9n, e portanto, n≥1/2. **Repare que,** para c=9, qualquer valor de m que escolhido acima de ½ estaria correto!

Exemplo: Comportamento Assintótico - Notação Grande-O

Seja um algoritmo cuja função de custo é $f(n)=3n+3$. Encontre as constantes positivas **c** e **m** para demonstrar que o algoritmo, no pior caso, tem o **limite assintótico superior** $O(n^2)$.

Exemplo: Comportamento Assintótico - Notação Grande-O

Seja um algoritmo cuja função de custo é f(n)=3n+3. Encontre as constantes positivas **c** e **m** para demonstrar que o algoritmo, no pior caso, tem o **limite assintótico superior** O(n^2).

Definição: Uma função de custo $f(n)$ é $O(g(n))$ se existem duas constantes positivas **c** e **m**, tais que, para $n \geq m$, temos $f(n) \leq cg(n)$.

- No nosso caso:
  - f(n)=3n+3
  - g(n)=n^2

Portanto:

- $f(n) \leq cg(n)$
- $3n+3 \leq cn^2$.

Vamos escolher c=9. Para a inequação $3n+3 \leq 9n^2$, podemos observar que ela é crescente em $n \geq 0.77$.

Logo, escolhendo m=0.77 e c=9 podemos dizer que $f(n) \leq cg(n)$, isto é, $3n+3 \leq cn^2$. Portanto existem duas constantes c e m que satisfazem a definição, e podemos dizer que a função de custo 3n+3 é O(n^2)

Atividade

1) Faça uma análise empírica da inserção no início e fim da lista sequencial estática e da inserção no início e fim da lista dinâmica encadeada, descritas nas implementações abaixo, considerando a inserção de até 100000 elementos:

Lista Sequencial Estática:http://www.facom.ufu.br/~backes/wordpress/ListaSequencial.zip

Lista dinâmica Encadeada:http://www.facom.ufu.br/~backes/wordpress/ListaDinamicaEncadeada.zip

- Dica (1): utilize a estratégia a seguir para medir o tempo: https://stackoverflow.com/questions/5248915/execution-time-of-c-program
- Dica (2): Tabele o tempo para as inserções de 10, 100, 1000… elementos.
- Dica (3): Como a análise empírica tem fatores externos que podem influenciar na mensuração do tempo, execute 10 vezes cada experimento e considere o tempo médio.

2) Analise as implementações do exercício anterior e responda: Qual é, no pior caso, o limite assintótico superior de ambas implementações?
3) Considere dois algoritmos A e B com funções de complexidade de tempo $a(n) = n^2 - n + 500$ e $b(n) = 47n + 47$, respectivamente. Para quais valores de n o algoritmo A leva menos tempo para executar do que B?
4) Indique se como verdadeiro ou falso as seguintes afirmações:
   a) $f(n)=2n+10$ é $O(n)$
   b) $f(n)=(3/2)n(n+1)$ é $O(n)$
   c) $f(n)=n/1000$ é $O(n)$
   d) $f(n)=2^{n+1}$ é $O(2^n)$

Comportamento Assintótico - notação *O - regra da soma*

- Se dois algoritmos são executados em sequência, a complexidade da execução dos dois algoritmos será dada pela complexidade do maior deles, ou seja:
  - $O(f(n)) + O(g(n)) = O(\max(f(n), g(n))$
- Por exemplo, se temos:
  - Dois algoritmos cujos tempos de execução são $O(n)$ e $O(n2)$, a execução deles em sequência será $O(\max(n, n2)$, que é $O(n2)$.
  - Dois algoritmos cujos tempos de execução são $O(n)$ e $O(n \log n)$, a execução deles em sequência será $O(\max(n, n \log n)$, que é $O(n \log n)$.

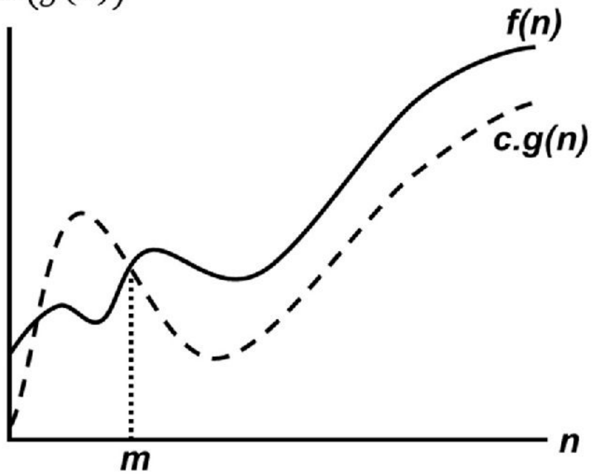
Comportamento Assintótico - Ω (lê-se grande-omega)

- A notação **Grande-O** é a mais utilizada, pois é o caso de mais fácil identificação (limite superior sobre o tempo de execução do algoritmo). Para diversos algoritmos, o pior caso ocorre com frequência.
- A notação Ω (lê-se grande-omega) descreve o **limite assintótico inferior** de um algoritmo. Trata-se de uma notação utilizada para analisar o **melhor caso** do algoritmo.
  - A notação $\Omega(n^2)$ nos diz que o custo do algoritmo é, assintoticamente, maior ou igual a n^2. Em outras palavras, o custo do algoritmo original é, no **mínimo**, tão ruim quanto n^2.

Comportamento Assintótico - notação Ω

- Matematicamente, a notação Ω é assim definida: uma função custo $f(n)$ é $\Omega(g(n))$ se existem duas constantes positivas **c** e **m**, tais que, para $n \geq m$, temos $f(n) \geq cg(n)$.
- Em outras palavras, para todos os valores de **n** à direita do valor **m**, o resultado da nossa função custo $f(n)$ é sempre **maior ou igual** ao valor da função usada na notação Ω, $g(n)$, multiplicada por uma constante **c**.

$$f(n) = \Omega\big(g(n)\big)$$



Exemplo: **Comportamento Assintótico -** notação Ω

Seja um algoritmo cuja função de custo é 3n+3. Encontre as constantes positivas **c** e **m** que provam que o algoritmo, no melhor caso, é Ω(n)

Exemplo: **Comportamento Assintótico -** notação Ω

Seja um algoritmo cuja função de custo é 3n+3. Encontre as constantes positivas **c** e **m** que provam que o algoritmo, no melhor caso, é Ω(n)

Definição: Uma função de custo $f(n)$ é $\Omega(g(n))$ se existem duas constantes positivas **c** e **m**, tais que, para $n \geq m$, temos $f(n) \geq cg(n)$.

- No nosso caso:
  - f(n)=3n+3
  - g(n)=n

Portanto:

- $f(n) \geq cg(n)$
- $3n+3 \geq cn$.

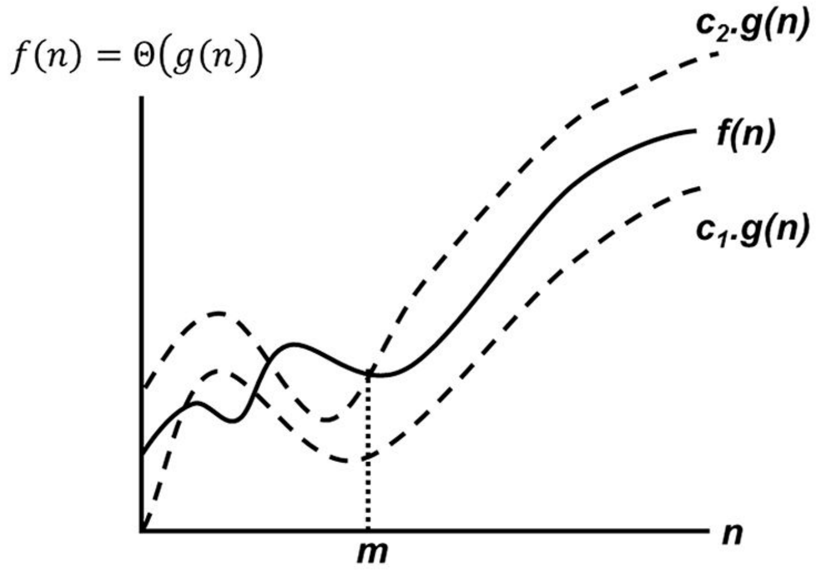
Vamos escolher c=2. Resolvendo a inequação, $3n+3 \geq 2n$, sabemos que ela é válida quando $n \geq -3$. Logo, para m=1 (constante positiva) e c=2 atendemos a definição. Portanto existem duas constantes c e m que satisfazem a definição, e podemos dizer que a função de custo 3n+3 é Ω(n)

Comportamento Assintótico - notação Θ

- A notação Θ (lê-se grande-theta) descreve o **limite assintótico firme** (ou **estreito**) de um algoritmo. Trata-se de uma notação utilizada para analisar o limite inferior e superior do algoritmo.
- A notação $\Theta(n^2)$ nos diz que o custo do algoritmo é, assintoticamente, igual a n^2. Em outras palavras, o custo do algoritmo original é n^2 dentro de um fator constante acima e abaixo.

Comportamento Assintótico - notação Θ

- Matematicamente, a notação Θ é assim definida: uma função custo $f(n)$ é $\Theta(g(n))$ se existem três constantes positivas $\mathbf{c_1}$, $\mathbf{c_2}$ e $\mathbf{m}$, tais que, para $n \geq m$, temos $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$.
- Em outras palavras, para todos os valores de **n** à direita do valor **m**, o resultado da nossa função custo $f(n)$ é sempre **igual** ao valor da função usada na notação Θ, $g(n)$, quando esta função é multiplicada por constantes $\mathbf{c_1}$ e $\mathbf{c_2}$.



Comportamento Assintótico - notação Θ

- Um exemplo simples desse tipo de notação consiste em mostrar que a seguinte função custo do nosso algoritmo:

$$f(n) = \frac{1}{2}n^2 - 3n$$

é $\Theta(n^2)$. Para tanto, iremos definir constantes positivas $\mathbf{c}_1$, $\mathbf{c}_2$ e $\mathbf{m}$, tais que:

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

para todo $n \geq m$. Dividindo por n^2 temos:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Comportamento Assintótico - notação Θ

- Conseguimos definir as constantes c_1, c_2 e n que satisfazem a inequação abaixo?

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

- Sim! $c_1 >= 1/14$, $c_2 >= 1/2$ e $n >= 7$
- Logo, podemos dizer que:

$$f(n) = \frac{1}{2}n^2 - 3n \quad \text{é} \quad \Theta(n^2).$$

Análise de Complexidade: Classes de Problemas

A seguir, são apresentadas algumas classes de complexidade de problemas comumente usadas:

- $O(1)$: **ordem constante**. As instruções são executadas um número fixo de vezes. Não depende do tamanho dos dados de entrada.
- $O(logN)$: **ordem logarítmica**. Típica de algoritmos que resolvem um problema transformando-o em problemas menores.
- $O(N)$: **ordem linear**. Em geral, certa quantidade de operações é realizada sobre cada um dos elementos de entrada.
- $O(NlogN)$: **ordem log linear**. Típica de algoritmos que trabalham com particionamento dos dados. Esses algoritmos resolvem um problema transformando-o em problemas menores, que são resolvidos de forma independente e depois unidos.
- $O(N^2)$: **ordem quadrática**. Normalmente, ocorre quando os dados são processados aos pares. Uma característica deste tipo de algoritmos é a presença de um aninhamento de dois comandos de repetição.
- $O(N^3)$: **ordem cúbica**. É caracterizado pela presença de três estruturas de repetição aninhadas.
- $O(2^N)$: **ordem exponencial**. Geralmente, ocorre quando se usa uma solução de **força bruta**. Não são úteis do ponto de vista prático.
- $O(N!)$: **ordem fatorial**. Geralmente, ocorre quando se usa uma solução de **força bruta**. Não são úteis do ponto de vista prático. Possui um comportamento muito pior que o exponencial.

Análise de Complexidade: Classes de Problemas

$f(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 50$	$n = 100$
n	1,0E-05 segundos	2,0E-05 segundos	4,0E-05 segundos	5,0E-05 segundos	6,0E-05 segundos
$n \log n$	3,3E-05 segundos	8,6E-05 segundos	2,1E-04 segundos	2,8E-04 segundos	3,5E-04 segundos
n^2	1,0E-04 segundos	4,0E-04 segundos	1,6E-03 segundos	2,5E-03 segundos	3,6E-03 segundos
n^3	1,0E-03 segundos	8,0E-03 segundos	6,4E-02 segundos	0,13 segundos	0,22 segundos
2^n	1,0E-03 segundos	1,0 segundo	2,8 dias	35,7 anos	365,6 séculos
3^n	5,9E-02 segundos	58,1 minutos	3855,2 séculos	2,3E+08 séculos	1,3E+13 séculos

Referências

Estrutura de Dados descomplicada em Linguagem C (André Backes): Cap 2;

Projeto de Algoritmos (Nivio Ziviani): Capítulo 1;